http://www.drdobbs.com/cpp/c-and-the-perils-of-double-checked-locki/184405726

# C++ singleton

Singleton in a single threaded environment is simple enough where you have a pointer pointing to null if the object is unallocated and pointing to the object otherwise.

// from the header file

class Singleton {

public:

static Singleton\* instance();

   ...

private:

   static Singleton\* pInstance;

};

// from the implementation file

Singleton\* Singleton::pInstance = 0;

Singleton\* Singleton::instance() {

   if (pInstance == 0) {

      pInstance = new Singleton;

   }

   return pInstance;

}

This implementation is not reliable in a multithreaded environment. Supposing a thread Thread A checked that the pointer is null and proceeded to allocate memory for the object. In the middle of allocating the object, it got swapped/preempted. Now ThreadB also finds that the pointer is null and proceeds to allocate memory and succeeds in creating the object and returns the object to the caller.

At some point later, Thread A is allowed to continue execution and it finished allocating where it was interrupted and returns the allocated object to its caller. So in effect we have two objects returned to two different callers for a singleton implementation which is contradictory is every sense.

Making it thread safe:

Making the implementation is easy, just take a lock in the function.

Singleton\* Singleton::instance() {

   Lock lock; // acquire lock (params omitted for simplicity)

   if (pInstance == 0) {

     pInstance = new Singleton;

   }

   return pInstance;

} // release lock (via Lock destructor)

The downside to this solution is that it may be expensive. Each access to the Singleton requires acquisition of a lock, but in reality, we need a lock only when initializing pInstance. That should occur only the first time instance is called. Ifinstance is called n times during the course of a program run, we need the lock only for the first call. Why pay for n lock acquisitions when you know that n-1 of them are unnecessary?

**The Double Checked Locking pattern**

DLCP tests pInstance in the usual way, however it takes a lock after it finds that the pInstance is NULL. After the lock is taken, pInstance is again checked to find whether it is null. The second test is necessary because it is possible that another thread happened to initialize pInstance between the time pInstance was first tested and the time the lock was acquired.

Singleton\* Singleton::instance() {

   if (pInstance == 0) { // 1st test

      Lock lock;

      if (pInstance == 0) { // 2nd test

         pInstance = new Singleton;

      }

   }

   return pInstance;

}

**DLCP and Instruction Ordering**

Consider the statement:

pInstance = new Singleton;

The above statement causes three things to happen:

Step 1: Allocate memory to hold a singleton Object.

Step 2: Construct a singleton object in the allocated memory

Step 3: Make pInstance point to the allocated memory

Now DLCP works very well if the above ordering is maintained.

However if the compiler reorder the instructions such that Step 3 is performed before Step 2, then there is an issue.

Given the above translation, consider the following sequence of events:

* Thread A enters instance, performs the first test of pInstance, acquires the lock, and executes the statement made up of Steps 1 and 3. It is then suspended. At this point,pInstance is not null, but no Singleton object has yet been constructed in the memorypInstance points to.
* Thread B enters instance, determines that pInstance is not null, and returns it toinstance's caller. The caller then dereferences the pointer to access the Singleton that, oops, has not yet been constructed.

DCLP works only if Steps 1 and 2 are completed before Step 3 is performed, but there is no way to express this constraint in C or C++. That's the dagger in the heart of DCLP—you need to define a constraint on relative instruction ordering, but the languages give you no way to express the constraint.

Why such a thing like above can happen:

Modern processors have a large word size and several execution units. Two or more arithmetic units are common. (For example, the Pentium 4 has three integer ALUs, PowerPC's G4e has four, and Itanium has six.) Their machine language allows compilers to generate code that yields parallel execution of two or more instructions in a single clock cycle.

Optimizing compilers carefully analyze and reorder your code so as to execute as many things at once as possible (within the constraints on observable behavior). Discovering and exploiting such parallelism in regular serial code is the single most important reason for rearranging code and introducing out-of-order execution. But it's not the only reason. Compilers (and linkers) might also reorder instructions to avoid spilling data from a register, to keep the instruction pipeline full, to perform common subexpression elimination, and reduce the size of the generated executable (see Bruno De Bus et al., "Post-pass Compaction Techniques").

When performing these kinds of optimizations, C/C++ compilers and linkers are constrained only by the dictates of observable behavior on the abstract machines defined by the language standards, and—this is the important bit—those abstract machines are implicitly single threaded. As languages, neither C nor C++ have threads, so compilers don't have to worry about breaking threaded programs when they are optimizing. It should, therefore, not surprise you that they sometimes do.

# C++ and the Perils of Double-Checked Locking: Part II

In the previous section, we examined why the Singleton pattern isn't thread safe, and how the Double-Checked Locking Pattern addresses that problem. This month, we look at the role the volatile keyword plays in this, and why DCLP may fail on both uni- and multiprocessor architectures.

The use of volatile as per the standards provides a sequence point wherein all side effects are expected to be handled when a sequence point is reached. So for a singleton, it would be a reasonable expectation to qualify the object and the pointer to be volatile so that multithreading problems with singleton implementation can be avoided.

class Singleton {

public:

static volatile Singleton\* volatile instance();

   ...

private:

   // one more volatile added

   static volatile Singleton\* volatile pInstance;

};

// from the implementation file

volatile Singleton\* volatile Singleton::pInstance = 0;

volatile Singleton\* volatile Singleton::instance() {

   if (pInstance == 0) {

      Lock lock;

      if (pInstance == 0) {

        // one more volatile added

        volatile Singleton\* volatile temp =

           new volatile Singleton;

        pInstance = temp;

      }

   }

   return pInstance;

}

The above is a glorified version of singleton where all the pawns are painted volatile.

This also may not work all the time and may fail for two reasons:

First, the Standard's constraints on observable behavior are only for an abstract machine defined by the Standard, and that abstract machine has no notion of multiple threads of execution. As a result, though the Standard prevents compilers from reordering reads and writes to volatile data within a thread, it imposes no constraints at all on such reorderings across threads. At least that's how most compiler implementers interpret things. As a result, in practice, many compilers may generate thread-unsafe code from the aforementioned source. If your multithreaded code works properly with volatile and doesn't work without it, then either your C++ implementation carefully implemented volatile to work with threads (less likely) or you simply got lucky (more likely). Either case, your code is not portable.

**DLCP on multiprocessor machines**

Suppose you're on a machine with multiple processors, each of which has its own memory cache, but all of which share a common memory space. Such an architecture needs to define exactly how and when writes performed by one processor propagate to the shared memory and thus become visible to other processors. It is easy to imagine situations where one processor has updated the value of a shared variable in its own cache, but the updated value has not yet been flushed to main memory, much less loaded into the other processors' caches. Such inter-cache inconsistencies in the value of a shared variable is known as the "cache coherency problem."

Suppose processor A modifies the memory for shared variable x and then later modifies the memory for shared variable y. These new values must be flushed to the main memory so that other processors see them. However, it can be more efficient to flush new cache values in increasing address order, so if y's address precedes x's, it is possible that y's new value will be written to main memory before x's is. If that happens, other processors may see y's value change before x's.

Such a possibility is a serious problem for DCLP. Correct Singleton initialization requires that the Singleton be initialized and that pInstance be updated to be non-null and that these operations be seen to occur in this order. If a thread on processor A performs step 1 and then step 2, but a thread on processor B sees step 2 as having been performed before step 1, the thread on processor B may again refer to an uninitialized Singleton

The general solution to cache coherency problems is to use memory barriers: instructions recognized by compilers, linkers, and other optimizing entities that constrain the kinds of reorderings that may be performed on read/writes of shared memory in multiprocessor systems. In the case of DCLP, we need to use memory barriers to ensure that pInstance isn't seen to be nonnull until writes to the Singleton have been completed.

Singleton\* Singleton::instance () {

   Singleton\* tmp = pInstance;

   ... // insert memory barrier

   if (tmp == 0) {

      Lock lock;

      tmp = pInstance;

      if (tmp == 0) {

         tmp = new Singleton;

         ... // insert memory barrier

         pInstance = tmp;

      }

   }

   return tmp;

}

This is a overkill but still this is an approach to implementing DCLP that should be reliable, provided you're running on a machine that supports memory barriers. All machines that can reorder writes to shared memory support memory barriers in one form or another. Interestingly, this same approach works just as well in a uniprocessor setting. This is because memory barriers also act as hard sequence points that prevent the kinds of instruction reorderings that can be so troublesome.

A good way to resolve the singleton problem is to use eager initialization; that is, to initialize a resource at the beginning of the program run. Because multithreaded programs typically start running as a single thread, this approach can push some object initializations into the single-threaded startup portion of the code, thus eliminating the need to worry about threading during the initialization. In many cases, initializing Singleton resource during single-threaded program startup (that is, prior to executing main) is the simplest way to offer fast, thread-safeSingleton access.

Finally, DCLP and its problems in C++ and C exemplify the inherent difficulty in writing thread-safe code in a language with no notion of threading (or any other form of concurrency). Multithreading considerations are pervasive because they affect the very core of code generation. As Peter Buhr pointed out in "Are Safe Concurrency Libraries Possible?" (see "References"), the desire to keep multi-threading out of the language and tucked away in libraries is a chimera. Do that, and either the libraries will end up putting constraints on the way compilers generate code (as Pthreads already does), or compilers and other code-generation tools will be prohibited from performing useful optimizations, even on single-threaded code. You can pick only two of the troika formed by multithreading, a thread-unaware language, and optimized code generation. Java and the .NET CLI, for example, address the tension by introducing thread awareness into the language and language infrastructure, respectively (see Doug Lea's *Concurrent Programming in Java*